3

Parsing

syn-tax: the way in which words are put together to form phrases, clauses, or sentences.

Webster's Dictionary

The abbreviation mechanism discussed in the previous chapter, whereby a symbol stands for some regular expression, is convenient enough that it is tempting to use it in interesting ways:

digits = [0 − 9]+
sum = (*digits* "+")* *digits*

These regular expressions define sums of the form `28+301+9`.

But now consider

digits = [0 − 9]+
sum = *expr* "+" *expr*
expr = "(" *sum* ")" | *digits*

This is meant to define expressions of the form:

```
(109+23)
61
(1+(250+3))
```

in which all the parentheses are balanced. But it is impossible for a finite automaton to recognize balanced parentheses (because a machine with *N* states cannot remember a parenthesis-nesting depth greater than *N*), so clearly *sum* and *expr* cannot be regular expressions.

So how does a lexical analyzer implement regular-expression abbreviations such as `digits`? The answer is that the right-hand-side `([0-9]+)` is

simply substituted for `digits` wherever it appears in regular expressions, *before* translation to a finite automaton.

This is not possible for the *sum*-and-*expr* language; we can first substitute *sum* into *expr*, yielding

expr = " (" *expr* "+" *expr* ") " | *digits*

but now an attempt to substitute *expr* into itself leads to

expr = " (" (" (" *expr* "+" *expr* ") " | *digits*) "+" *expr* ") " | *digits*

and the right-hand side now has just as many occurrences of *expr* as it did before – in fact, it has more!

Thus, the notion of abbreviation does not add expressive power to the language of regular expressions – there are no additional languages that can be defined – unless the abbreviations are recursive (or mutually recursive, as are *sum* and *expr*).

The additional expressive power gained by recursion is just what we need for parsing. Also, once we have abbreviations with recursion, we do not need alternation except at the top level of expressions, because the definition

$$expr = ab(c \mid d)e$$

can always be rewritten using an auxiliary definition as

$$aux = c \mid d$$
$$expr = a\ b\ aux\ e$$

In fact, instead of using the alternation mark at all, we can just write several allowable expansions for the same symbol:

$$aux = c$$
$$aux = d$$
$$expr = a\ b\ aux\ e$$

The Kleene closure is not necessary, since we can rewrite it so that

$$expr = (a\ b\ c)*$$

becomes

$$expr = (a\ b\ c)\ expr$$
$$expr = \epsilon$$

1 $S \rightarrow S\ ;\ S$
2 $S \rightarrow \text{id} := E$
3 $S \rightarrow \text{print}\ (\ L\ )$
4 $E \rightarrow \text{id}$
5 $E \rightarrow \text{num}$
6 $E \rightarrow E\ +\ E$
7 $E \rightarrow (S\ ,\ E\ )$
8 $L \rightarrow E$
9 $L \rightarrow L\ ,\ E$

GRAMMAR 3.1. A syntax for straight-line programs.

What we have left is a very simple notation, called *context-free grammars*. Just as regular expressions can be used to define lexical structure in a static, declarative way, grammars define syntactic structure declaratively. But we will need something more powerful than finite automata to parse languages described by grammars.

In fact, grammars can also be used to describe the structure of lexical tokens, although regular expressions are adequate – and more concise – for that purpose.

3.1 CONTEXT-FREE GRAMMARS

As before, we say that a *language* is a set of *strings*; each string is a finite sequence of *symbols* taken from a finite *alphabet*. For parsing, the strings are source programs, the symbols are lexical tokens, and the alphabet is the set of token-types returned by the lexical analyzer.

A context-free grammar describes a language. A grammar has a set of *productions* of the form

$$symbol \rightarrow symbol\ symbol \cdots symbol$$

where there are zero or more symbols on the right-hand side. Each symbol is either *terminal*, meaning that it is a token from the alphabet of strings in the language, or *nonterminal*, meaning that it appears on the left-hand side of some production. No token can ever appear on the left-hand side of a production. Finally, one of the nonterminals is distinguished as the *start symbol* of the grammar.

Grammar 3.1 is an example of a grammar for straight-line programs. The start symbol is S (when the start symbol is not written explicitly it is conventional to assume that the left-hand nonterminal in the first production is the start symbol). The terminal symbols are

id print num , + ( ) := ;

$\underline{S}$
S ; $\underline{S}$
$\underline{S}$; id := E
id := $\underline{E}$; id := E
id := num ; id := $\underline{E}$
id := num ; id := E + $\underline{E}$
id := num ; id := $\underline{E}$ + (S , E)
id := num ; id := id + ($\underline{S}$, E)
id := num ; id := id + (id := $\underline{E}$, E)
id := num ; id := id + (id := E + E , $\underline{E}$)
id := num ; id := id + (id := $\underline{E}$ + E , id)
id := num ; id := id + (id := num + $\underline{E}$, id)
id := num ; id := id + (id := num + num , id)

DERIVATION 3.2.

and the nonterminals are S, E, and L. One sentence in the language of this grammar is

```
id := num; id := id + (id := num + num, id)
```

where the source text (before lexical analysis) might have been

```
a := 7;
b := c + (d := 5 + 6, d)
```

The token-types (terminal symbols) are `id`, `num`, `:=`, and so on; the names (`a,b,c,d`) and numbers (7, 5, 6) are *semantic values* associated with some of the tokens.

DERIVATIONS

To show that this sentence is in the language of the grammar, we can perform a *derivation*: Start with the start symbol, then repeatedly replace any nonterminal by one of its right-hand sides, as shown in Derivation 3.2.

There are many different derivations of the same sentence. A *leftmost* derivation is one in which the leftmost nonterminal symbol is always the one expanded; in a *rightmost* derivation, the rightmost nonterminal is always the next to be expanded.

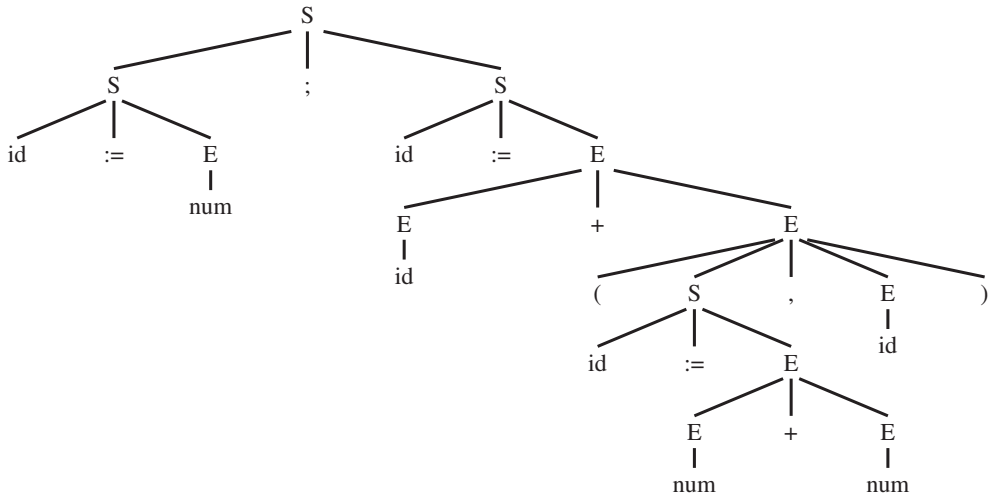


FIGURE 3.3. Parse tree.

Derivation 3.2 is neither leftmost nor rightmost; a leftmost derivation for this sentence would begin,

$$
\begin{aligned}
&\underline{S}\\
&\underline{S}\ ;\ S\\
&\mathrm{id}\ :=\ \underline{E}\ ;\ S\\
&\mathrm{id}\ :=\ \mathrm{num}\ ;\ \underline{S}\\
&\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \underline{E}\\
&\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \underline{E}\ +\ E\\
&\quad\vdots
\end{aligned}
$$

PARSE TREES

A *parse tree* is made by connecting each symbol in a derivation to the one from which it was derived, as shown in Figure 3.3. Two different derivations can have the same parse tree.

AMBIGUOUS GRAMMARS

A grammar is *ambiguous* if it can derive a sentence with two different parse trees. Grammar 3.1 is ambiguous, since the sentence `id := id+id+id` has two parse trees (Figure 3.4).

Grammar 3.5 is also ambiguous; Figure 3.6 shows two parse trees for the sentence `1-2-3`, and Figure 3.7 shows two trees for `1+2*3`. Clearly, if we use

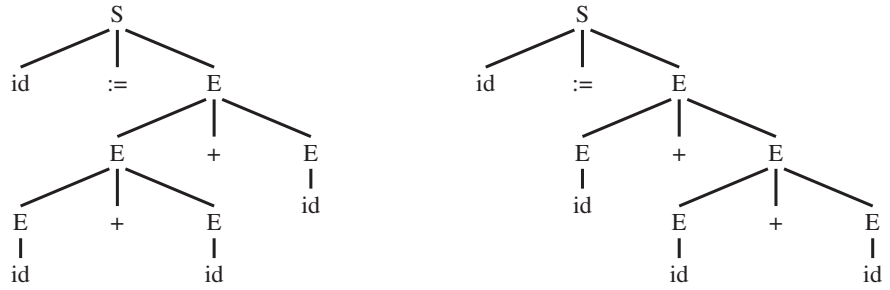


FIGURE 3.4. Two parse trees for the same sentence using Grammar 3.1.

$E \rightarrow \text{id}$
$E \rightarrow \text{num}$
$E \rightarrow E * E$
$E \rightarrow E / E$
$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow (E)$

GRAMMAR 3.5.

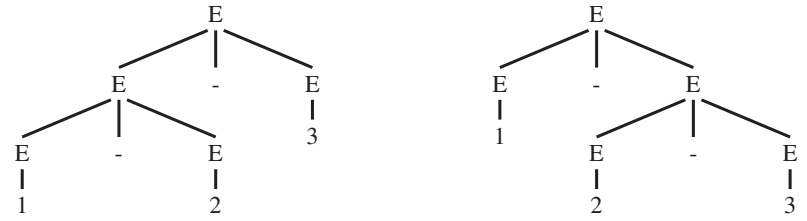


FIGURE 3.6. Two parse trees for the sentence `1-2-3` in Grammar 3.5.

FIGURE 3.7. Two parse trees for the sentence `1+2*3` in Grammar 3.5.

$E \to E + T$	$T \to T * F$	$F \to \text{id}$
$E \to E - T$	$T \to T / F$	$F \to \text{num}$
$E \to T$	$T \to F$	$F \to (E)$

GRAMMAR 3.8.

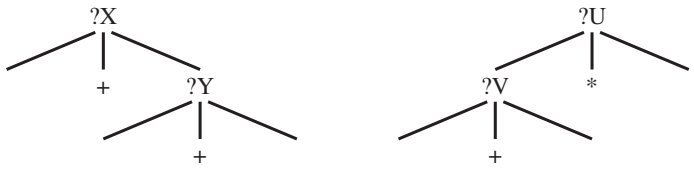


FIGURE 3.9. Parse trees that Grammar 3.8 will never produce.

parse trees to interpret the meaning of the expressions, the two parse trees for `1-2-3` mean different things: $(1 - 2) - 3 = -4$ versus $1 - (2 - 3) = 2$. Similarly, $(1 + 2) \times 3$ is not the same as $1 + (2 \times 3)$. And indeed, compilers do use parse trees to derive meaning.

Therefore, ambiguous grammars are problematic for compiling: In general, we would prefer to have unambiguous grammars. Fortunately, we can often transform ambiguous grammars to unambiguous grammars.

Let us find an unambiguous grammar that accepts the same language as Grammar 3.5. First, we would like to say that `*` *binds tighter* than `+`, or has *higher precedence*. Second, we want to say that each operator *associates to the left*, so that we get $(1 - 2) - 3$ instead of $1 - (2 - 3)$. We do this by introducing new nonterminal symbols to get Grammar 3.8.

The symbols E, T, and F stand for *expression*, *term*, and *factor*; conventionally, factors are things you multiply and terms are things you add.

This grammar accepts the same set of sentences as the ambiguous grammar, but now each sentence has exactly one parse tree. Grammar 3.8 can never produce parse trees of the form shown in Figure 3.9 (see Exercise 3.17).

Had we wanted to make `*` associate to the right, we could have written its production as $T \to F * T$.

We can usually eliminate ambiguity by transforming the grammar. Though there are some languages (sets of strings) that have ambiguous grammars but no unambiguous grammar, such languages may be problematic as *programming* languages because the syntactic ambiguity may lead to problems in writing and understanding programs.

$S \to E\ \$$

$E \to E + T$
$E \to E - T$
$E \to T$

$T \to T * F$
$T \to T / F$
$T \to F$

$F \to \text{id}$
$F \to \text{num}$
$F \to (\ E\ )$

GRAMMAR 3.10.

$S \to \text{if } E \text{ then } S \text{ else } S$
$S \to \text{begin } S\ L$
$S \to \text{print } E$

$L \to \text{end}$
$L \to ;\ S\ L$

$E \to \text{num} = \text{num}$

GRAMMAR 3.11.

END-OF-FILE MARKER

Parsers must read not only terminal symbols such as `+`, `-`, `num`, and so on, but also the end-of-file marker. We will use $ to represent end of file.

Suppose S is the start symbol of a grammar. To indicate that $ must come after a complete S-phrase, we augment the grammar with a new start symbol S' and a new production $S' \to S\$$.

In Grammar 3.8, E is the start symbol, so an augmented grammar is Grammar 3.10.

3.2 PREDICTIVE PARSING

Some grammars are easy to parse using a simple algorithm known as *recursive descent*. In essence, each grammar production turns into one clause of a recursive function. We illustrate this by writing a recursive-descent parser for Grammar 3.11.

A recursive-descent parser for this language has one function for each nonterminal and one clause for each production.

```
final int IF=1, THEN=2, ELSE=3, BEGIN=4, END=5, PRINT=6,
          SEMI=7, NUM=8, EQ=9;

int tok = getToken();

void advance() {tok=getToken();}
void eat(int t) {if (tok==t) advance(); else error();}

void S() {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();
                    eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();
       }}
void L() {switch(tok) {
        case END:   eat(END); break;
        case SEMI:  eat(SEMI); S(); L(); break;
        default:    error();
       }}
void E() {  eat(NUM); eat(EQ); eat(NUM); }
```

With suitable definitions of `error` and `getToken`, this program will parse very nicely.

Emboldened by success with this simple method, let us try it with Grammar 3.10:

```
void S() {  E(); eat(EOF); }
void E() {switch (tok) {
          case ?: E(); eat(PLUS); T(); break;
          case ?: E(); eat(MINUS); T(); break;
          case ?: T(); break;
          default: error();
         }}
void T() {switch (tok) {
          case ?: T(); eat(TIMES); F(); break;
          case ?: T(); eat(DIV); F(); break;
          case ?: F(); break;
          default: error();
         }}
```

There is a *conflict* here: The E function has no way to know which clause to use. Consider the strings `(1*2-3)+4` and `(1*2-3)`. In the former case, the initial call to E should use the $E \rightarrow E + T$ production, but the latter case should use $E \rightarrow T$.

$Z \to d$	$Y \to$	$X \to Y$
$Z \to X\ Y\ Z$	$Y \to c$	$X \to a$

GRAMMAR 3.12.

Recursive-descent, or *predictive*, parsing works only on grammars where the *first terminal symbol* of each subexpression provides enough information to choose which production to use. To understand this better, we will formalize the notion of FIRST sets, and then derive conflict-free recursive-descent parsers using a simple algorithm.

Just as lexical analyzers can be constructed from regular expressions, there are parser-generator tools that build predictive parsers. But if we are going to use a tool, then we might as well use one based on the more powerful LR(1) parsing algorithm, which will be described in Section 3.3.

Sometimes it's inconvenient or impossible to use a parser-generator tool. The advantage of predictive parsing is that the algorithm is simple enough that we can use it to construct parsers by hand – we don't need automatic tools.

FIRST AND FOLLOW SETS

Given a string γ of terminal and nonterminal symbols, FIRST(γ) is the set of all terminal symbols that can begin any string derived from γ. For example, let $\gamma = T * F$. Any string of terminal symbols derived from γ must start with `id`, `num`, or `(`. Thus, FIRST($T * F$) = {`id`, `num`, `(`}.

If two different productions $X \to \gamma_1$ and $X \to \gamma_2$ have the same left-hand-side symbol (X) and their right-hand sides have overlapping FIRST sets, then the grammar cannot be parsed using predictive parsing. If some terminal symbol I is in FIRST(γ_1) and also in FIRST(γ_2), then the X function in a recursive-descent parser will not know what to do if the input token is I.

The computation of FIRST sets looks very simple: If $\gamma = X\,Y\,Z$, it seems as if Y and Z can be ignored, and FIRST(X) is the only thing that matters. But consider Grammar 3.12. Because Y can produce the empty string – and therefore X can produce the empty string – we find that FIRST($X\,Y\,Z$) must include FIRST(Z). Therefore, in computing FIRST sets, we must keep track of which symbols can produce the empty string; we say such symbols are *nullable*. And we must keep track of what might follow a nullable symbol.

With respect to a particular grammar, given a string γ of terminals and nonterminals,

- nullable(X) is true if X can derive the empty string.
- FIRST(γ) is the set of terminals that can begin strings derived from γ.
- FOLLOW(X) is the set of terminals that can immediately follow X. That is, $t \in$ FOLLOW(X) if there is any derivation containing Xt. This can occur if the derivation contains $X\,Y\,Z\,t$ where Y and Z both derive ϵ.

A precise definition of FIRST, FOLLOW, and nullable is that they are the smallest sets for which these properties hold:

For each terminal symbol Z, FIRST$[Z] = \{Z\}$.
for each production $X \to Y_1 Y_2 \cdots Y_k$
  if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$)
    then nullable$[X]$ = true
  for each i from 1 to k, each j from $i + 1$ to k
    if $Y_1 \cdots Y_{i-1}$ are all nullable (or if $i = 1$)
      then FIRST$[X]$ = FIRST$[X] \cup$ FIRST$[Y_i]$
    if $Y_{i+1} \cdots Y_k$ are all nullable (or if $i = k$)
      then FOLLOW$[Y_i]$ = FOLLOW$[Y_i] \cup$ FOLLOW$[X]$
    if $Y_{i+1} \cdots Y_{j-1}$ are all nullable (or if $i + 1 = j$)
      then FOLLOW$[Y_i]$ = FOLLOW$[Y_i] \cup$ FIRST$[Y_j]$

Algorithm 3.13 for computing FIRST, FOLLOW, and nullable just follows from these facts; we simply replace each equation with an assignment statement, and iterate.

Of course, to make this algorithm efficient it helps to examine the productions in the right order; see Section 17.4. Also, the three relations need not be computed simultaneously; nullable can be computed by itself, then FIRST, then FOLLOW.

This is not the first time that a group of equations on sets has become the algorithm for calculating those sets; recall the algorithm on page 28 for computing ϵ-closure. Nor will it be the last time; the technique of iteration to a fixed point is applicable in dataflow analysis for optimization, in the back end of a compiler.

We can apply this algorithm to Grammar 3.12. Initially, we have:

	nullable	FIRST	FOLLOW
X	no		
Y	no		
Z	no		

Algorithm to compute FIRST, FOLLOW, *and* nullable.
Initialize FIRST and FOLLOW to all empty sets, and nullable to all false.
for each terminal symbol Z
    FIRST$[Z] \leftarrow \{Z\}$
repeat
    for each production $X \rightarrow Y_1 Y_2 \cdots Y_k$
        if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$)
          then nullable$[X] \leftarrow$ true
        for each i from 1 to k, each j from $i + 1$ to k
            if $Y_1 \cdots Y_{i-1}$ are all nullable (or if $i = 1$)
              then FIRST$[X] \leftarrow$ FIRST$[X] \cup$ FIRST$[Y_i]$
            if $Y_{i+1} \cdots Y_k$ are all nullable (or if $i = k$)
              then FOLLOW$[Y_i] \leftarrow$ FOLLOW$[Y_i] \cup$ FOLLOW$[X]$
            if $Y_{i+1} \cdots Y_{j-1}$ are all nullable (or if $i + 1 = j$)
              then FOLLOW$[Y_i] \leftarrow$ FOLLOW$[Y_i] \cup$ FIRST$[Y_j]$
until FIRST, FOLLOW, and nullable did not change in this iteration.

ALGORITHM 3.13. Iterative computation of *FIRST, FOLLOW,* and *nullable*.

In the first iteration, we find that $a \in$ FIRST$[X]$, Y is nullable, $c \in$ FIRST$[Y]$, $d \in$ FIRST$[Z]$, $d \in$ FOLLOW$[X]$, $c \in$ FOLLOW$[X]$, $d \in$ FOLLOW$[Y]$. Thus:

	nullable	FIRST	FOLLOW
X	no	a	c d
Y	yes	c	d
Z	no	d	

In the second iteration, we find that X is nullable, $c \in$ FIRST$[X]$, $\{a, c\} \subseteq$ FIRST$[Z]$, $\{a, c, d\} \subseteq$ FOLLOW$[X]$, $\{a, c, d\} \subseteq$ FOLLOW$[Y]$. Thus:

	nullable	FIRST	FOLLOW
X	yes	a c	a c d
Y	yes	c	a c d
Z	no	a c d	

The third iteration finds no new information, and the algorithm terminates.

	a	c	d
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$

FIGURE 3.14. Predictive parsing table for Grammar 3.12.

It is useful to generalize the FIRST relation to strings of symbols:

$$\begin{aligned} &\text{FIRST}(X\gamma) = \text{FIRST}[X] && \text{if not nullable}[X] \\ &\text{FIRST}(X\gamma) = \text{FIRST}[X] \cup \text{FIRST}(\gamma) && \text{if nullable}[X] \end{aligned}$$

and similarly, we say that a string γ is nullable if each symbol in γ is nullable.

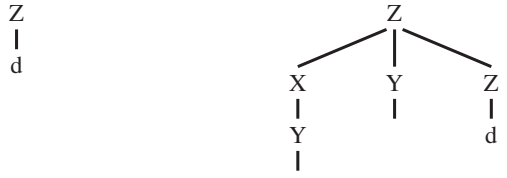
CONSTRUCTING A PREDICTIVE PARSER

Consider a recursive-descent parser. The parsing function for some nonterminal X has a clause for each X production; it must choose one of these clauses based on the next token T of the input. If we can choose the right production for each (X, T), then we can write the recursive-descent parser. All the information we need can be encoded as a two-dimensional table of productions, indexed by nonterminals X and terminals T. This is called a *predictive parsing* table.

To construct this table, enter production $X \to \gamma$ in row X, column T of the table for each $T \in \text{FIRST}(\gamma)$. Also, if γ is nullable, enter the production in row X, column T for each $T \in \text{FOLLOW}[X]$.

Figure 3.14 shows the predictive parser for Grammar 3.12. But some of the entries contain more than one production! The presence of duplicate entries means that predictive parsing will not work on Grammar 3.12.

If we examine the grammar more closely, we find that it is ambiguous. The sentence d has many parse trees, including:



An ambiguous grammar will always lead to duplicate entries in a predictive parsing table. If we need to use the language of Grammar 3.12 as a programming language, we will need to find an unambiguous grammar.

Grammars whose predictive parsing tables contain no duplicate entries are called LL(1). This stands for *left-to-right parse, leftmost-derivation, 1-symbol lookahead*. Clearly a recursive-descent (predictive) parser examines the input left-to-right in one pass (some parsing algorithms do not, but these are generally not useful for compilers). The order in which a predictive parser expands nonterminals into right-hand sides (that is, the recursive-descent parser calls functions corresponding to nonterminals) is just the order in which a leftmost derivation expands nonterminals. And a recursive-descent parser does its job just by looking at the next token of the input, never looking more than one token ahead.

We can generalize the notion of FIRST sets to describe the first k tokens of a string, and to make an LL(k) parsing table whose rows are the nonterminals and columns are every sequence of k terminals. This is rarely done (because the tables are so large), but sometimes when you write a recursive-descent parser by hand you need to look more than one token ahead.

Grammars parsable with LL(2) parsing tables are called LL(2) grammars, and similarly for LL(3), etc. Every LL(1) grammar is an LL(2) grammar, and so on. No ambiguous grammar is LL(k) for any k.

ELIMINATING LEFT RECURSION

Suppose we want to build a predictive parser for Grammar 3.10. The two productions

$$E \to E + T$$
$$E \to T$$

are certain to cause duplicate entries in the LL(1) parsing table, since any token in FIRST(T) will also be in FIRST($E + T$). The problem is that E appears as the first right-hand-side symbol in an E-production; this is called *left recursion*. Grammars with left recursion cannot be LL(1).

To eliminate left recursion, we will rewrite using right recursion. We introduce a new nonterminal E', and write

$$E \to T\ E'$$

$$E' \to +\ T\ E'$$
$$E' \to$$

$$
\begin{array}{lll}
S \rightarrow E\ \$ & & \\
 & T \rightarrow F\ T' & \\
E \rightarrow T\ E' & & F \rightarrow \text{id} \\
 & T' \rightarrow *\ F\ T' & F \rightarrow \text{num} \\
E' \rightarrow +\ T\ E' & T' \rightarrow /\ F\ T' & F \rightarrow (\ E\ ) \\
E' \rightarrow -\ T\ E' & T' \rightarrow & \\
E' \rightarrow & &
\end{array}
$$

GRAMMAR 3.15.

	nullable	FIRST	FOLLOW
S	no	(id num	
E	no	(id num	) $
E'	yes	+ -	) $
T	no	(id num	) + - $
T'	yes	* /	) + - $
F	no	(id num	) * / + - $

TABLE 3.16. Nullable, FIRST, and FOLLOW for Grammar 3.15.

This derives the same set of strings (on T and $+$) as the original two productions, but now there is no left recursion.

In general, whenever we have productions $X \rightarrow X\gamma$ and $X \rightarrow \alpha$, where α does not start with X, we know that this derives strings of the form $\alpha\gamma*$, an α followed by zero or more γ. So we can rewrite the regular expression using right recursion:

$$
\begin{pmatrix}
X \rightarrow X\ \gamma_1 \\
X \rightarrow X\ \gamma_2 \\
X \rightarrow \alpha_1 \\
X \rightarrow \alpha_2
\end{pmatrix}
\Longrightarrow
\begin{pmatrix}
X \rightarrow \alpha_1\ X' \\
X \rightarrow \alpha_2\ X' \\
X' \rightarrow \gamma_1\ X' \\
X' \rightarrow \gamma_2\ X' \\
X' \rightarrow
\end{pmatrix}
$$

Applying this transformation to Grammar 3.10, we obtain Grammar 3.15.

To build a predictive parser, first we compute nullable, FIRST, and FOLLOW (Table 3.16). The predictive parser for Grammar 3.15 is shown in Table 3.17.

	+	*	id	(	)	$
S			$S \to E\$$	$S \to E\$$		
E			$E \to TE'$	$E \to TE'$		
E'	$E' \to +TE'$				$E' \to$	$E' \to$
T			$T \to FT'$	$T \to FT'$		
T'	$T' \to$	$T' \to *FT'$			$T' \to$	$T' \to$
F			$F \to$ id	$F \to (E)$		

TABLE 3.17. Predictive parsing table for Grammar 3.15. We omit the columns for `num`, `/`, and `-`, as they are similar to others in the table.

LEFT FACTORING

We have seen that left recursion interferes with predictive parsing, and that it can be eliminated. A similar problem occurs when two productions for the same nonterminal start with the same symbols. For example:

$S \to$ if E then S else S
$S \to$ if E then S

In such a case, we can *left factor* the grammar – that is, take the allowable endings (*else* S and ϵ) and make a new nonterminal X to stand for them:

$S \to$ if E then S X
$X \to$
$X \to$ else S

The resulting productions will not pose a problem for a predictive parser. Although the grammar is still ambiguous – the parsing table has two entries for the same slot – we can resolve the ambiguity by using the *else* S action.

ERROR RECOVERY

Armed with a predictive parsing table, it is easy to write a recursive-descent parser. Here is a representative fragment of a parser for Grammar 3.15:

```
void T() {switch (tok) {
        case ID:
        case NUM:
        case LPAREN:  F(); Tprime(); break;
        default:   error!
      }}

void Tprime() {switch (tok) {
        case PLUS:    break;
        case TIMES:   eat(TIMES); F(); Tprime(); break;
        case EOF:     break;
        case RPAREN:  break;
        default:   error!
      }}
```

A blank entry in row T, column x of the LL(1) parsing table indicates that the parsing function `T()` does not expect to see token x – this will be a syntax error. How should *error* be handled? It is safe just to raise an exception and quit parsing, but this is not very friendly to the user. It is better to print an error message and recover from the error, so that other syntax errors can be found in the same compilation.

A syntax error occurs when the string of input tokens is not a sentence in the language. Error recovery is a way of finding some sentence similar to that string of tokens. This can proceed by deleting, replacing, or inserting tokens.

For example, error recovery for `T` could proceed by inserting a `num` token. It's not necessary to adjust the actual input; it suffices to pretend that the `num` was there, print a message, and return normally.

```
void T() {switch (tok) {
        case ID:
        case NUM:
        case LPAREN: F(); Tprime(); break;
        default:  print("expected id, num, or left-paren");
      }}
```

It's a bit dangerous to do error recovery by insertion, because if the error cascades to produce another error, the process might loop infinitely. Error recovery by deletion is safer, because the loop must eventually terminate when end-of-file is reached.

Simple recovery by deletion works by skipping tokens until a token in the FOLLOW set is reached. For example, error recovery for T' could work like this:

```
int Tprime_follow [] = {PLUS, RPAREN, EOF};

void Tprime() { switch (tok) {
        case PLUS:   break;
        case TIMES:  eat(TIMES); F(); Tprime(); break;
        case RPAREN: break;
        case EOF:    break;
        default:   print("expected +, *, right-paren,
                          or end-of-file");
                   skipto(Tprime_follow);
       }}
```

A recursive-descent parser's error-recovery mechanisms must be adjusted (sometimes by trial and error) to avoid a long cascade of error-repair messages resulting from a single token out of place.

3.3 LR PARSING

The weakness of LL(k) parsing techniques is that they must *predict* which production to use, having seen only the first k tokens of the right-hand side. A more powerful technique, LR(k) parsing, is able to postpone the decision until it has seen input tokens corresponding to the entire right-hand side of the production in question (and k more input tokens beyond).

LR(k) stands for *left-to-right parse, rightmost-derivation, k-token lookahead.* The use of a rightmost derivation seems odd; how is that compatible with a left-to-right parse? Figure 3.18 illustrates an LR parse of the program

```
a := 7;
b := c + (d := 5 + 6, d)
```

using Grammar 3.1, augmented with a new start production $S' \rightarrow S\$$.

The parser has a *stack* and an *input*. The first k tokens of the input are the *lookahead*. Based on the contents of the stack and the lookahead, the parser performs two kinds of actions:

Shift: Move the first input token to the top of the stack.
Reduce: Choose a grammar rule $X \rightarrow A\ B\ C$; pop C, B, A from the top of the stack; push X onto the stack.

Initially, the stack is empty and the parser is at the beginning of the input. The action of shifting the end-of-file marker \$ is called *accepting* and causes the parser to stop successfully.

Stack	*Input*	*Action*
$_1$	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ id$_4$	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ id$_4$:=$_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ id$_4$:=$_6$ num$_{10}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num
$_1$ id$_4$:=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ id:=E
$_1$ S_2	; b := c + (d := 5 + 6 , d) $	*shift*
$_1$ S_2 ;$_3$	b := c + (d := 5 + 6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$	:= c + (d := 5 + 6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$	c + (d := 5 + 6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ id$_{20}$	+ (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ id
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11}	+ (d := 5 + 6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$	(d := 5 + 6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$	d := 5 + 6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$	:= 5 + 6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$	5 + 6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ num$_{10}$	+ 6 , d) $	*reduce* $E \rightarrow$ num
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11}	+ 6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11} +$_{16}$	6 , d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11} +$_{16}$ num$_{10}$	, d) $	*reduce* $E \rightarrow$ num
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	, d) $	*reduce* $E \rightarrow E + E$
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ id$_4$:=$_6$ E_{11}	, d) $	*reduce* $S \rightarrow$ id:=E
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12}	, d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12} ,$_{18}$	d) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12} ,$_{18}$ id$_{20}$	) $	*reduce* $E \rightarrow$ id
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12} ,$_{18}$ E_{21}	) $	*shift*
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ ($_8$ S_{12} ,$_{18}$ E_{21})$_{22}$	$	*reduce* $E \rightarrow (S, E)$
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	$	*reduce* $E \rightarrow E + E$
$_1$ S_2 ;$_3$ id$_4$:=$_6$ E_{11}	$	*reduce* $S \rightarrow$ id:=E
$_1$ S_2 ;$_3$ S_5	$	*reduce* $S \rightarrow S; S$
$_1$ S_2	$	*accept*

FIGURE 3.18. Shift-reduce parse of a sentence. Numeric subscripts in the *Stack* are DFA state numbers; see Table 3.19.

In Figure 3.18, the stack and input are shown after every step, along with an indication of which action has just been performed. The concatenation of stack and input is always one line of a rightmost derivation; in fact, Figure 3.18 shows the rightmost derivation of the input string, upside-down.

LR PARSING ENGINE

How does the LR parser know when to shift and when to reduce? By using a deterministic finite automaton! The DFA is not applied to the input – finite automata are too weak to parse context-free grammars – but to the stack. The edges of the DFA are labeled by the symbols (terminals and non-

	id	num	print	;	,	+	:=	(	)	$	*S*	*E*	*L*
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9	s20	s10						s8				g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

TABLE 3.19. LR parsing table for Grammar 3.1.

terminals) that can appear on the stack. Table 3.19 is the transition table for Grammar 3.1.

The elements in the transition table are labeled with four kinds of actions:

s*n*	Shift into state *n*;
g*n*	Goto state *n*;
r*k*	Reduce by rule *k*;
a	Accept;
	Error (denoted by a blank entry in the table).

To use this table in parsing, treat the shift and goto actions as edges of a DFA, and scan the stack. For example, if the stack is id := *E*, then the DFA goes from state 1 to 4 to 6 to 11. If the next input token is a semicolon, then the ";" column in state 11 says to reduce by rule 2. The second rule of the grammar is $S \rightarrow \text{id} := E$, so the top three tokens are popped from the stack and *S* is pushed.

The action for "+" in state 11 is to shift; so if the next token had been + instead, it would have been eaten from the input and pushed on the stack.

0 $S' \rightarrow S\$$

1 $S \rightarrow (L)$

2 $S \rightarrow x$

3 $L \rightarrow S$

4 $L \rightarrow L , S$

GRAMMAR 3.20.

Rather than rescan the stack for each token, the parser can remember instead the state reached for each stack element. Then the parsing algorithm is

Look up top stack state, and input symbol, to get action;
If action is
Shift(n): Advance input one token; push n on stack.
Reduce(k): Pop stack as many times as the number of symbols on the right-hand side of rule k;
Let X be the left-hand-side symbol of rule k;
In the state now on top of stack, look up X to get "goto n";
Push n on top of stack.
Accept: Stop parsing, report success.
Error: Stop parsing, report failure.

LR(0) PARSER GENERATION

An LR(k) parser uses the contents of its stack and the next k tokens of the input to decide which action to take. Table 3.19 shows the use of one symbol of lookahead. For $k = 2$, the table has columns for every two-token sequence and so on; in practice, $k > 1$ is not used for compilation. This is partly because the tables would be huge, but more because most reasonable programming languages can be described by $LR(1)$ grammars.

LR(0) grammars are those that can be parsed looking only at the stack, making shift/reduce decisions without any lookahead. Though this class of grammars is too weak to be very useful, the algorithm for constructing LR(0) parsing tables is a good introduction to the LR(1) parser construction algorithm.

We will use Grammar 3.20 to illustrate LR(0) parser generation. Consider what the parser for this grammar will be doing. Initially, it will have an empty stack, and the input will be a complete S-sentence followed by \$; that is, the right-hand side of the S' rule will be on the input. We indicate this as $S' \rightarrow .S\$$ where the dot indicates the current position of the parser.

In this state, where the input begins with S, that means that it begins with any possible right-hand side of an S-production; we indicate that by

$$\boxed{\begin{array}{l} S' \to .S\$ \\ S \to .x \\ S \to .(L) \end{array}}^{1}$$

Call this state 1. A grammar rule, combined with the dot that indicates a position in its right-hand side, is called an *item* (specifically, an *LR(0) item*). A state is just a set of items.

Shift actions. In state 1, consider what happens if we shift an x. We then know that the end of the stack has an x; we indicate that by shifting the dot past the x in the $S \to x$ production. The rules $S' \to .S\$$ and $S \to .(L)$ are irrelevant to this action, so we ignore them; we end up in state 2:

$$\boxed{S \to x.}^{2}$$

Or in state 1 consider shifting a left parenthesis. Moving the dot past the parenthesis in the third item yields $S \to (.L)$, where we know that there must be a left parenthesis on top of the stack, and the input begins with some string derived by L, followed by a right parenthesis. What tokens can begin the input now? We find out by including all L-productions in the set of items. But now, in one of those L-items, the dot is just before an S, so we need to include all the S-productions:

$$\boxed{\begin{array}{l} S \to (.L) \\ L \to .L, S \\ L \to .S \\ S \to .(L) \\ S \to .x \end{array}}^{3}$$

Goto actions. In state 1, consider the effect of parsing past some string of tokens derived by the S nonterminal. This will happen when an x or left parenthesis is shifted, followed (eventually) by a reduction of an S-production. All the right-hand-side symbols of that production will be popped, and the parser will execute the goto action for S in state 1. The effect of this can be simulated by moving the dot past the S in the first item of state 1, yielding state 4:

$$\boxed{S' \to S.\$}^{4}$$

Reduce actions. In state 2 we find the dot at the end of an item. This means that on top of the stack there must be a complete right-hand side of the corresponding production ($S \rightarrow x$), ready to reduce. In such a state the parser could perform a reduce action.

The basic operations we have been performing on states are **closure**(I) and **goto**(I, X), where I is a set of items and X is a grammar symbol (terminal or nonterminal). **Closure** adds more items to a set of items when there is a dot to the left of a nonterminal; **goto** moves the dot past the symbol X in all items.

```
Closure(I) =
  repeat
    for any item A → α.Xβ in I
      for any production X → γ
        I ← I ∪ {X → .γ}
  until I does not change.
  return I
```

```
Goto(I, X) =
  set J to the empty set
  for any item A → α.Xβ in I
    add A → αX.β to J
  return Closure(J)
```

Now here is the algorithm for LR(0) parser construction. First, augment the grammar with an auxiliary start production $S' \rightarrow S\$$. Let T be the set of states seen so far, and E the set of (shift or goto) edges found so far.

```
Initialize T to {Closure({S' → .S$})}
Initialize E to empty.
repeat
    for each state I in T
      for each item A → α.Xβ in I
        let J be Goto(I, X)
        T ← T ∪ {J}
        E ← E ∪ {I -X-> J}
until E and T did not change in this iteration
```

However, for the symbol \$ we do not compute **Goto**(I, \$); instead we will make an **accept** action.

For Grammar 3.20 this is illustrated in Figure 3.21.

Now we can compute set R of LR(0) reduce actions:

```
R ← {}
for each state I in T
  for each item A → α. in I
    R ← R ∪ {(I, A → α)}
```

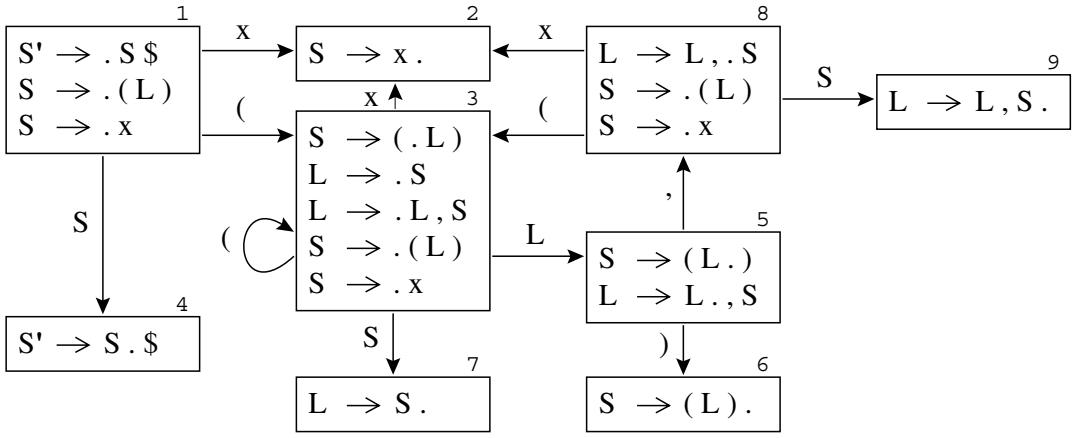


FIGURE 3.21. LR(0) states for Grammar 3.20.

	(	)	x	,	$	*S*	*L*
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

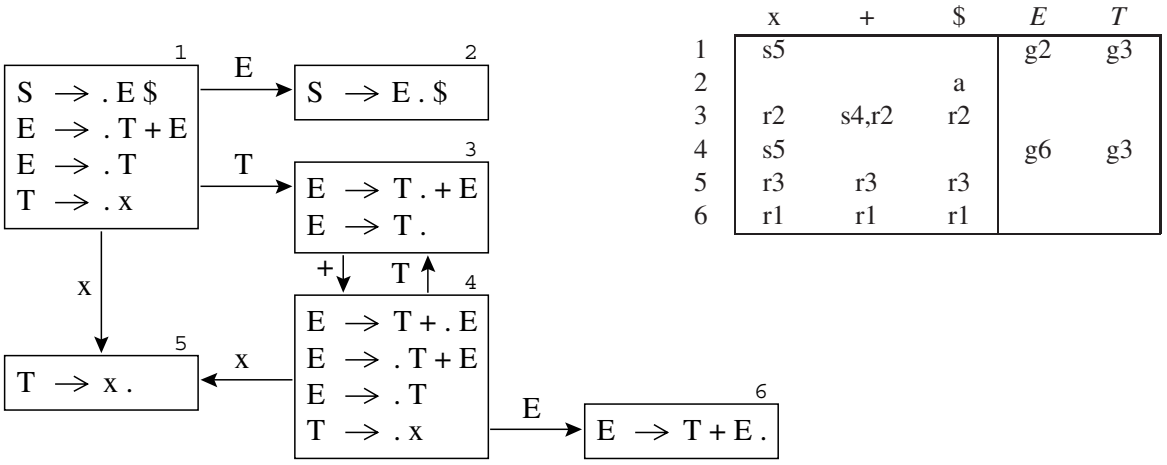
TABLE 3.22. LR(0) parsing table for Grammar 3.20.

We can now construct a parsing table for this grammar (Table 3.22). For each edge $I \xrightarrow{X} J$ where X is a terminal, we put the action *shift* J at position (I, X) of the table; if X is a nonterminal, we put *goto* J at position (I, X). For each state I containing an item $S' \rightarrow S.\$$ we put an *accept* action at $(I, \$)$. Finally, for a state containing an item $A \rightarrow \gamma.$ (production n with the dot at the end), we put a *reduce* n action at (I, Y) for every token Y.

In principle, since LR(0) needs no lookahead, we just need a single action for each state: A state will shift or reduce, but not both. In practice, since we need to know what state to shift into, we have rows headed by state numbers and columns headed by grammar symbols.

0 $S \rightarrow E\ \$$
1 $E \rightarrow T\ +\ E$
2 $E \rightarrow T$
3 $T \rightarrow x$

GRAMMAR 3.23.



	x	+	$	*E*	*T*
1	s5			g2	g3
2			a		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

FIGURE 3.24. LR(0) states and parsing table for Grammar 3.23.

SLR PARSER GENERATION

Let us attempt to build an LR(0) parsing table for Grammar 3.23. The LR(0) states and parsing table are shown in Figure 3.24.

In state 3, on symbol +, there is a duplicate entry: The parser must shift into state 4 and also reduce by production 2. This is a conflict and indicates that the grammar is not LR(0) – it cannot be parsed by an LR(0) parser. We will need a more powerful parsing algorithm.

A simple way of constructing better-than-LR(0) parsers is called SLR, which stands for simple LR. Parser construction for SLR is almost identical to that for LR(0), except that we put reduce actions into the table only where indicated by the FOLLOW set.

Here is the algorithm for putting reduce actions into an SLR table:

$R \leftarrow \{\}$
for each state I in T
  for each item $A \rightarrow \alpha.$ in I
    for each token X in FOLLOW(A)
      $R \leftarrow R \cup \{(I,\ X,\ A \rightarrow \alpha)\}$

	x	+	$	*E*	*T*
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

FIGURE 3.25. SLR parsing table for Grammar 3.23.

The action $(I, X, A \rightarrow \alpha)$ indicates that in state I, on lookahead symbol X, the parser will reduce by rule $A \rightarrow \alpha$.

Thus, for Grammar 3.23 we use the same LR(0) state diagram (Figure 3.24), but we put fewer reduce actions into the SLR table, as shown in Figure 3.25.

The SLR class of grammars is precisely those grammars whose SLR parsing table contains no conflicts (duplicate entries). Grammar 3.23 belongs to this class, as do many useful programming-language grammars.

LR(1) ITEMS; LR(1) PARSING TABLE

Even more powerful than SLR is the LR(1) parsing algorithm. Most programming languages whose syntax is describable by a context-free grammar have an LR(1) grammar.

The algorithm for constructing an LR(1) parsing table is similar to that for LR(0), but the notion of an *item* is more sophisticated. An LR(1) item consists of a *grammar production*, a *right-hand-side position* (represented by the dot), and a *lookahead symbol*. The idea is that an item $(A \rightarrow \alpha.\beta,\ x)$ indicates that the sequence α is on top of the stack, and at the head of the input is a string derivable from βx.

An LR(1) state is a set of LR(1) items, and there are **Closure** and **Goto** operations for LR(1) that incorporate the lookahead:

Closure(I) =
  repeat
    for any item $(A \rightarrow \alpha.X\beta, z)$ in I
      for any production $X \rightarrow \gamma$
        for any $w \in$ FIRST(βz)
          $I \leftarrow I \cup \{(X \rightarrow .\gamma,\ w)\}$
  until I does not change
  return I

Goto(I, X) =
  $J \leftarrow \{\}$
  for any item $(A \rightarrow \alpha.X\beta,\ z)$ in I
    add $(A \rightarrow \alpha X.\beta,\ z)$ to J
  return Closure(J).

The start state is the closure of the item ($S' \rightarrow .S\ \$,\ ?$), where the lookahead symbol ? will not matter, because the end-of-file marker will never be shifted.

The reduce actions are chosen by this algorithm:

$R \leftarrow \{\}$
for each state I in T
  for each item $(A \rightarrow \alpha.\ ,\ z)$ in I
    $R \leftarrow R \cup \{(I, z, A \rightarrow \alpha)\}$

The action $(I, z, A \rightarrow \alpha)$ indicates that in state I, on lookahead symbol z, the parser will reduce by rule $A \rightarrow \alpha$.

Grammar 3.26 is not SLR (see Exercise 3.9), but it is in the class of LR(1) grammars. Figure 3.27 shows the LR(1) states for this grammar; in the figure, where there are several items with the same production but different lookahead, as at left below, we have abbreviated as at right:

$S' \rightarrow .\ S\ \$$	?
$S \rightarrow .\ V = E$	$
$S \rightarrow .\ E$	$
$E \rightarrow .\ V$	$
$V \rightarrow .$ x	$
$V \rightarrow .\ * E$	$
$V \rightarrow .$ x	=
$V \rightarrow .\ * E$	=

$S' \rightarrow .\ S\ \$$	?
$S \rightarrow .\ V = E$	$
$S \rightarrow .\ E$	$
$E \rightarrow .\ V$	$
$V \rightarrow .$ x	$, =
$V \rightarrow .\ * E$	$, =

The LR(1) parsing table derived from this state graph is Table 3.28a. Wherever the dot is at the end of a production (as in state 3 of Figure 3.27, where it is at the end of production $E \rightarrow V$), then there is a *reduce* action for that production in the LR(1) table, in the row corresponding to the state number and the column corresponding to the lookahead of the item (in this case, the lookahead is $). Whenever the dot is to the left of a terminal symbol or nonterminal, there is a corresponding shift or goto action in the LR(1) parsing table, just as there would be in an LR(0) table.

LALR(1) PARSING TABLES

LR(1) parsing tables can be very large, with many states. A smaller table can be made by merging any two states whose items are identical except for lookahead sets. The result parser is called an LALR(1) parser, for *lookahead LR(1)*.

0 $S' \rightarrow S\,\$$
1 $S \rightarrow V = E$
2 $S \rightarrow E$
3 $E \rightarrow V$
4 $V \rightarrow$ x
5 $V \rightarrow * E$

GRAMMAR 3.26. A grammar capturing the essence of expressions, variables, and pointer-dereference (by the *) operator in the C language.

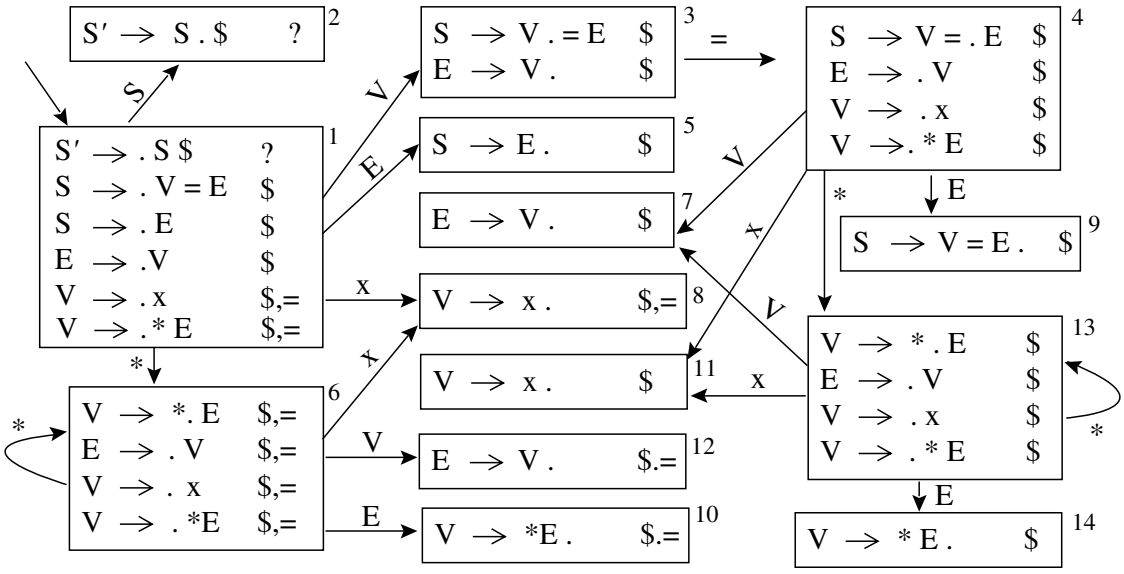


FIGURE 3.27. LR(1) states for Grammar 3.26.

	x	*	=	$	*S*	*E*	*V*
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

(a) LR(1)

	x	*	=	$	*S*	*E*	*V*
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s8	s6				g9	g7
5				r2			
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

(b) LALR(1)

TABLE 3.28. LR(1) and LALR(1) parsing tables for Grammar 3.26.

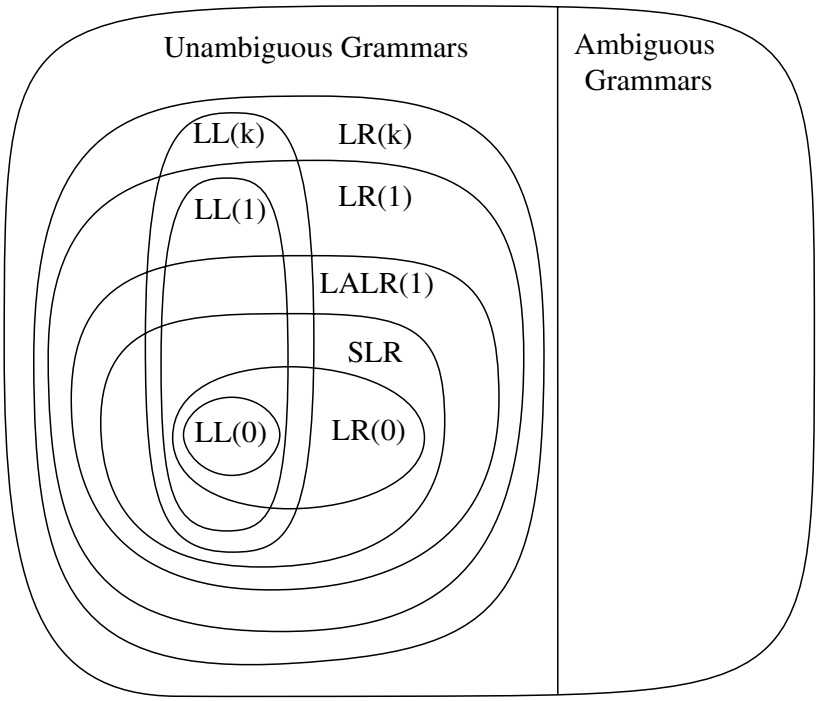


FIGURE 3.29. A hierarchy of grammar classes.

For example, the items in states 6 and 13 of the LR(1) parser for Grammar 3.26 (Figure 3.27) are identical if the lookahead sets are ignored. Also, states 7 and 12 are identical except for lookahead, as are states 8 and 11 and states 10 and 14. Merging these pairs of states gives the LALR(1) parsing table shown in Table 3.28b.

For some grammars, the LALR(1) table contains reduce-reduce conflicts where the LR(1) table has none, but in practice the difference matters little. What does matter is that the LALR(1) parsing table requires less memory to represent than the LR(1) table, since there can be many fewer states.

HIERARCHY OF GRAMMAR CLASSES

A grammar is said to be LALR(1) if its LALR(1) parsing table contains no conflicts. All SLR grammars are LALR(1), but not vice versa. Figure 3.29 shows the relationship between several classes of grammars.

Any reasonable programming language has a LALR(1) grammar, and there are many parser-generator tools available for LALR(1) grammars. For this

reason, LALR(1) has become a standard for programming languages and for automatic parser generators.

LR PARSING OF AMBIGUOUS GRAMMARS

Many programming languages have grammar rules such as

$S \rightarrow$ if E then S else S
$S \rightarrow$ if E then S
$S \rightarrow$ other

which allow programs such as

```
if a then if b then s1 else s2
```

Such a program could be understood in two ways:

```
(1)        if a then { if b then s1 else s2 }
(2)        if a then { if b then s1 } else s2
```

In most programming languages, an `else` must match the most recent possible `then`, so interpretation *(1)* is correct. In the LR parsing table there will be a shift-reduce conflict:

$S \rightarrow$ if E then S .	else
$S \rightarrow$ if E then S . else S	(*any*)

Shifting corresponds to interpretation *(1)* and reducing to interpretation *(2)*.

The ambiguity can be eliminated by introducing auxiliary nonterminals M (for *matched statement*) and U (for *unmatched statement*):

$S \rightarrow M$
$S \rightarrow U$
$M \rightarrow$ if E then M else M
$M \rightarrow$ other
$U \rightarrow$ if E then S
$U \rightarrow$ if E then M else U

But instead of rewriting the grammar, we can leave the grammar unchanged and tolerate the shift-reduce conflict. In constructing the parsing table this conflict should be resolved by shifting, since we prefer interpretation *(1)*.

1 $P \rightarrow L$

2 $S \rightarrow$ id := id
3 $S \rightarrow$ while id do S
4 $S \rightarrow$ begin L end
5 $S \rightarrow$ if id then S
6 $S \rightarrow$ if id then S else S
7 $L \rightarrow S$
8 $L \rightarrow L$; S

GRAMMAR 3.30.

It is often possible to use ambiguous grammars by resolving shift-reduce conflicts in favor of shifting or reducing, as appropriate. But it is best to use this technique sparingly, and only in cases (such as the *dangling-else* described here, and operator-precedence to be described on page 74) that are well understood. Most shift-reduce conflicts, and probably all reduce-reduce conflicts, should not be resolved by fiddling with the parsing table. They are symptoms of an ill-specified grammar, and they should be resolved by eliminating ambiguities.

3.4 USING PARSER GENERATORS

The task of constructing a parser is simple enough to be automated. In the previous chapter we described the lexical-analyzer aspects of JavaCC and SableCC. Here we will discuss the parser-generator aspects of these tools. Documentation for JavaCC and SableCC are available via this book's Web site.

JAVACC

JavaCC is an LL(k) parser generator. Productions are of the form:

```
void Assignment() : {} { Identifier() "=" Expression() ";" }
```

where the left-hand side is `Assignment()`; the right-hand side is enclosed between the last two curly brackets; `Assignment()`, `Identifier()`, and `Expression()` are nonterminal symbols; and `"="` and `";"` are terminal symbols.

Grammar 3.30 can be represented as a JavaCC grammar as shown in Gram-

```
PARSER_BEGIN(MyParser)
  public class MyParser {}
PARSER_END(MyParser)

SKIP :
{ " " | "\t" | "\n" }

TOKEN :
{ < WHILE: "while" >
| < BEGIN: "begin" >
| < END: "end" >
| < DO: "do" >
| < IF: "if" >
| < THEN: "then" >
| < ELSE: "else" >
| < SEMI: ";" >
| < ASSIGN: "=" >
| < ID: ["a"-"z"](["a"-"z"] | ["0"-"9"])* >
}

void Prog() :
{}
{ StmList() <EOF> }

void StmList() :
{}
{ Stm() StmListPrime() }

void StmListPrime() :
{}
{ ( ";" Stm() StmListPrime() )? }

void Stm() :
{}
{ <ID> "=" <ID>
| "while" <ID> "do" Stm()
| "begin" StmList() "end"
| LOOKAHEAD(5)  /* we need to lookahead till we see "else" */
  "if" <ID> "then" Stm()
| "if" <ID> "then" Stm() "else" Stm()
}
```

GRAMMAR 3.31. JavaCC version of Grammar 3.30.

mar 3.31. Notice that if we had written the production for `StmList()` in the style of Grammar 3.30, that is,

```
void StmList() :
{}
{ Stm()
| StmList( ) ";" Stm()
}
```

then the grammar would be left recursive. In that case, JavaCC would give the following error:

```
Left recursion detected: "StmList... --> StmList..."
```

We used the techniques mentioned earlier to remove the left recursion and arrive at Grammar 3.31.

SABLECC

SableCC is an LALR(1) parser generator. Productions are of the form:

```
assignment = identifier assign expression semicolon ;
```

where the left-hand side is `assignment`; the right-hand side is enclosed between `=` and `;`; `assignment`, `identifier`, and `expression` are nonterminal symbols; and `assign` and `semicolon` are terminal symbols that are defined in an earlier part of the syntax specification.

Grammar 3.30 can be represented as a SableCC grammar as shown in Grammar 3.32. When there is more than one alternative, SableCC requires a name for each alternative. A name is given to an alternative in the grammar by prefixing the alternative with an identifier between curly brackets. Also, if the same grammar symbol appears twice in the same alternative of a production, SableCC requires a name for at least one of the two elements. Element names are specified by prefixing the element with an identifier between square brackets followed by a colon.

SableCC reports shift-reduce and reduce-reduce conflicts. A shift-reduce conflict is a choice between shifting and reducing; a reduce-reduce conflict is a choice of reducing by two different rules.

SableCC will report that the Grammar 3.32 has a shift-reduce conflict. The conflict can be examined by reading the detailed error message SableCC produces, as shown in Figure 3.33.

```
Tokens
    while = 'while';
    begin = 'begin';
    end = 'end';
    do = 'do';
    if = 'if';
    then = 'then';
    else = 'else';
    semi = ';';
    assign = '=';
    whitespace = (' ' | '\t' | '\n')+;
    id = ['a'..'z'](['a'..'z'] | ['0'..'9'])*;
Ignored Tokens
    whitespace;
Productions
    prog = stmlist;

    stm =  {assign} [left]:id assign [right]:id |
           {while} while id do stm |
           {begin} begin stmlist end |
           {if_then} if id then stm |
           {if_then_else} if id then [true_stm]:stm else [false_stm]:stm;

    stmlist = {stmt} stm |
              {stmtlist} stmlist semi stm;
```

GRAMMAR 3.32. SableCC version of Grammar 3.30.

```
shift/reduce conflict in state [stack: TIf TId TThen PStm *] on TElse in {
        [ PStm = TIf TId TThen PStm * TElse PStm ] (shift),
        [ PStm = TIf TId TThen PStm * ] followed by TElse (reduce)
}
```

FIGURE 3.33. SableCC shift-reduce error message for Grammar 3.32.

SableCC prefixes productions with an uppercase 'P' and tokens with an uppercase 'T', and replaces the first letter with an uppercase when it makes the objects for the tokens and productions. This is what you see on the stack in the error message in Figure 3.33. So on the stack we have tokens for `if`, `id`, `then`, and a production that matches a `stm`, and now we have an `else` token. Clearly this reveals that the conflict is caused by the familiar dangling `else`.

In order to resolve this conflict we need to rewrite the grammar, removing the ambiguity as in Grammar 3.34.

```
Productions
    prog = stmlist;

    stm =  {stm_without_trailing_substm}
               stm_without_trailing_substm |
           {while} while id do stm |
           {if_then} if id then stm |
           {if_then_else} if id then stm_no_short_if
                          else [false_stm]:stm;

    stm_no_short_if = {stm_without_trailing_substm}
                          stm_without_trailing_substm |
                      {while_no_short_if}
                          while id do stm_no_short_if |
                     {if_then_else_no_short_if}
                        if id then [true_stm]:stm_no_short_if
                              else [fals_stm]:stm_no_short_if;

    stm_without_trailing_substm  =  {assign} [left]:id assign [right]:id |
                                       {begin} begin stmlist end ;

    stmlist = {stmt} stm | {stmtlist} stmlist semi stm;
```

GRAMMAR 3.34. SableCC productions of Grammar 3.32 with conflicts resolved.

PRECEDENCE DIRECTIVES

No ambiguous grammar is LR(k) for any k; the LR(k) parsing table of an ambiguous grammar will always have conflicts. However, ambiguous grammars can still be useful if we can find ways to resolve the conflicts.

For example, Grammar 3.5 is highly ambiguous. In using this grammar to describe a programming language, we intend it to be parsed so that $*$ and / bind more tightly than $+$ and $-$, and that each operator associates to the left. We can express this by rewriting the unambiguous Grammar 3.8.

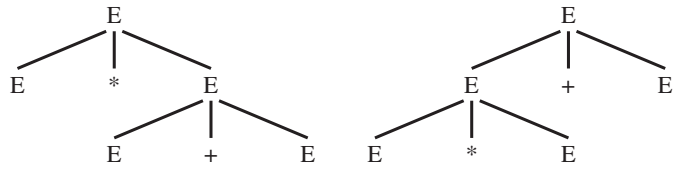
But we can avoid introducing the T and F symbols and their associated "trivial" reductions $E \rightarrow T$ and $T \rightarrow F$. Instead, let us start by building the LR(1) parsing table for Grammar 3.5, as shown in Table 3.35. We find many conflicts. For example, in state 13 with lookahead $+$ we find a conflict between *shift into state 8* and *reduce by rule 3*. Two of the items in state 13 are

$E \rightarrow E * E .$	$+$
$E \rightarrow E . + E$	*(any)*

	id	num	+	-	*	/	(	)	$	*E*
1	s2	s3					s4			g7
2			r1	r1	r1	r1		r1	r1	
3			r2	r2	r2	r2		r2	r2	
4	s2	s3					s4			g5
5								s6		
6			r7	r7	r7	r7		r7	r7	
7			s8	s10	s12	s14			a	
8	s2	s3					s4			g9
9			s8,r5	s10,r5	s12,r5	s14,r5		r5	r5	
10	s2	s3					s4			g11
11			s8,r6	s10,r6	s12,r6	s14,r6		r6	r6	
12	s2	s3					s4			g13
13			s8,r3	s10,r3	s12,r3	s14,r3		r3	r3	
14	s2	s3					s4			g15
15			s8,r4	s10,r4	s12,r4	s14,r4		r4	r4	

TABLE 3.35. LR parsing table for Grammar 3.5.

In this state the top of the stack is $\cdots E * E$. Shifting will lead to a stack $\cdots E * E+$ and eventually $\cdots E * E + E$ with a reduction of $E + E$ to E. Reducing now will lead to the stack $\cdots E$ and then the $+$ will be shifted. The parse trees obtained by shifting and reducing are



Shift Reduce

If we wish $*$ to bind tighter than $+$, we should reduce instead of shift. So we fill the $(13, +)$ entry in the table with r3 and discard the s8 action.

Conversely, in state 9 on lookahead $*$, we should shift instead of reduce, so we resolve the conflict by filling the $(9, *)$ entry with s12.

The case for state 9, lookahead $+$ is

$E \to E + E .$	$+$
$E \to E . + E$	*(any)*

Shifting will make the operator right-associative; reducing will make it left-associative. Since we want left associativity, we fill $(9, +)$ with r5.

Consider the expression $a - b - c$. In most programming languages, this

	+	-	*	/	
			⋮		
9		r5	r5	s12	s14	
11	···			s12	s14	···
13		r3	r3	r3	r3	
15		r4	r4			
			⋮		

TABLE 3.36. Conflicts of Table 3.35 resolved.

associates to the left, as if written $(a - b) - c$. But suppose we believe that this expression is inherently confusing, and we want to force the programmer to put in explicit parentheses, either $(a - b) - c$ or $a - (b - c)$. Then we say that the minus operator is *nonassociative*, and we would fill the $(11, -)$ entry with an error entry.

The result of all these decisions is a parsing table with all conflicts resolved (Table 3.36).

Yacc has *precedence directives* to indicate the resolution of this class of shift-reduce conflicts. (Unfortunately, SableCC does not have precedence directives.) A series of declarations such as

```
precedence nonassoc EQ, NEQ;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
precedence right EXP;
```

indicates that + and - are left-associative and bind equally tightly; that * and / are left-associative and bind more tightly than +; that ^ is right-associative and binds most tightly; and that = and $\neq$ are nonassociative, and bind more weakly than +.

In examining a shift-reduce conflict such as

$E \to E * E .$	$+$
$E \to E . + E$	*(any)*

there is the choice of shifting a *token* and reducing by a *rule*. Should the rule or the token be given higher priority? The precedence declarations (`precedence left`, etc.) give priorities to the tokens; the priority of a rule is given by the last token occurring on the right-hand side of that rule. Thus the choice here is between a rule with priority * and a token with priority +; the rule has higher priority, so the conflict is resolved in favor of reducing.

```
%{  declarations of yylex and yyerror %}
%token INT PLUS MINUS TIMES UMINUS
%start exp

%left PLUS MINUS
%left TIMES
%left UMINUS
%%

exp :  INT
    |  exp PLUS exp
    |  exp MINUS exp
    |  exp TIMES exp
    |  MINUS exp    %prec UMINUS
```

GRAMMAR 3.37. Yacc grammar with precedence directives.

When the rule and token have equal priority, then a `left` precedence favors reducing, `right` favors shifting, and `nonassoc` yields an error action.

Instead of using the default "rule has precedence of its last token," we can assign a specific precedence to a rule using the `%prec` directive. This is commonly used to solve the "unary minus" problem. In most programming languages a unary minus binds tighter than any binary operator, so $-6 * 8$ is parsed as $(-6) * 8$, not $-(6 * 8)$. Grammar 3.37 shows an example.

The token UMINUS is never returned by the lexer; it's just a placeholder in the chain of `precedence` declarations. The directive `%prec UMINUS` gives the rule `exp::= MINUS exp` the highest precedence, so reducing by this rule takes precedence over shifting any operator, even a minus sign.

Precedence rules are helpful in resolving conflicts, but they should not be abused. If you have trouble explaining the effect of a clever use of precedence rules, perhaps instead you should rewrite the grammar to be unambiguous.

SYNTAX VERSUS SEMANTICS

Consider a programming language with *arithmetic expressions* such as $x + y$ and *boolean expressions* such as $x + y = z$ or $a \& (b = c)$. Arithmetic operators bind tighter than the boolean operators; there are arithmetic variables and boolean variables; and a boolean expression cannot be added to an arithmetic expression. Grammar 3.38 gives a syntax for this language.

The grammar has a reduce-reduce conflict. How should we rewrite the grammar to eliminate this conflict?

Here the problem is that when the parser sees an identifier such as a, it has

```
%token ID ASSIGN PLUS MINUS AND EQUAL
%start stm
%left OR
%left AND
%left PLUS
%%

stm : ID ASSIGN ae
    | ID ASSIGN be

be  : be OR be
    | be AND be
    | ae EQUAL ae
    | ID

ae  : ae PLUS ae
    | ID
```

GRAMMAR 3.38. Yacc grammar with precedence directives.

no way of knowing whether this is an arithmetic variable or a boolean variable – syntactically they look identical. The solution is to defer this analysis until the "semantic" phase of the compiler; it's not a problem that can be handled naturally with context-free grammars. A more appropriate grammar is

$$S \rightarrow \text{id} := E$$

$$E \rightarrow \text{id}$$
$$E \rightarrow E \,\&\, E$$
$$E \rightarrow E = E$$
$$E \rightarrow E + E$$

Now the expression $a + 5\&b$ is syntactically legal, and a later phase of the compiler will have to reject it and print a semantic error message.

3.5 ERROR RECOVERY

LR(k) parsing tables contain shift, reduce, accept, and error actions. On page 58 we claimed that when an LR parser encounters an error action it stops parsing and reports failure. This behavior would be unkind to the programmer, who would like to have *all* the errors in her program reported, not just the first error.

RECOVERY USING THE ERROR SYMBOL

Local error recovery mechanisms work by adjusting the parse stack and the input *at the point where the error was detected* in a way that will allow parsing to resume. One local recovery mechanism – found in many versions of the Yacc parser generator – uses a special *error* symbol to control the recovery process. Wherever the special *error* symbol appears in a grammar rule, a sequence of erroneous input tokens can be matched.

For example, in a Yacc grammar we might have productions such as

$$exp \rightarrow ID$$
$$exp \rightarrow exp \ + \ exp$$
$$exp \rightarrow (\ exps\ )$$
$$exps \rightarrow exp$$
$$exps \rightarrow exps\ ;\ exp$$

Informally, we can specify that if a syntax error is encountered in the middle of an expression, the parser should skip to the next semicolon or right parenthesis (these are called *synchronizing tokens*) and resume parsing. We do this by adding error-recovery productions such as

$$exp \rightarrow (\ error\ )$$
$$exps \rightarrow error\ ;\ exp$$

What does the parser generator do with the *error* symbol? In parser generation, *error* is considered a terminal symbol, and shift actions are entered in the parsing table for it as if it were an ordinary token.

When the LR parser reaches an error state, it takes the following actions:

1. Pop the stack (if necessary) until a state is reached in which the action for the *error* token is *shift*.
2. Shift the *error* token.
3. Discard input symbols (if necessary) until a lookahead is reached that has a nonerror action in the current state.
4. Resume normal parsing.

In the two *error* productions illustrated above, we have taken care to follow the *error* symbol with an appropriate synchronizing token – in this case, a right parenthesis or semicolon. Thus, the "nonerror action" taken in step 3 will always *shift*. If instead we used the production $exp \rightarrow error$, the "nonerror action" would be *reduce*, and (in an SLR or LALR parser) it is possible that the original (erroneous) lookahead symbol would cause another error after the reduce action, without having advanced the input. Therefore, grammar

rules that contain *error* not followed by a token should be used only when there is no good alternative.

Caution. One can attach *semantic actions* to Yacc grammar rules; whenever a rule is reduced, its semantic action is executed. Chapter 4 explains the use of semantic actions. Popping states from the stack can lead to seemingly "impossible" semantic actions, especially if the actions contain side effects. Consider this grammar fragment:

```
statements:  statements exp SEMICOLON
           |  statements error SEMICOLON
           |  /* empty */

exp : increment exp decrement
    | ID

increment:  LPAREN     {: nest=nest+1; :}
decrement:  RPAREN     {: nest=nest-1; :}
```

"Obviously" it is true that whenever a semicolon is reached, the value of `nest` is zero, because it is incremented and decremented in a balanced way according to the grammar of expressions. But if a syntax error is found after some left parentheses have been parsed, then states will be popped from the stack without "completing" them, leading to a nonzero value of `nest`. The best solution to this problem is to have side-effect-free semantic actions that build abstract syntax trees, as described in Chapter 4.

Unfortunately, neither JavaCC nor SableCC support the *error*-symbol error-recovery method, nor the kind of global error repair described below.

GLOBAL ERROR REPAIR

Global error repair finds the smallest set of insertions and deletions that would turn the source string into a syntactically correct string, *even if the insertions and deletions are not at a point where an LL or LR parser would first report an error*.

Burke-Fisher error repair. We will describe a limited but useful form of global error repair, which tries every possible single-token insertion, deletion, or replacement at every point that occurs no earlier than K tokens before the point where the parser reported the error. Thus, with $K = 15$, if the parsing

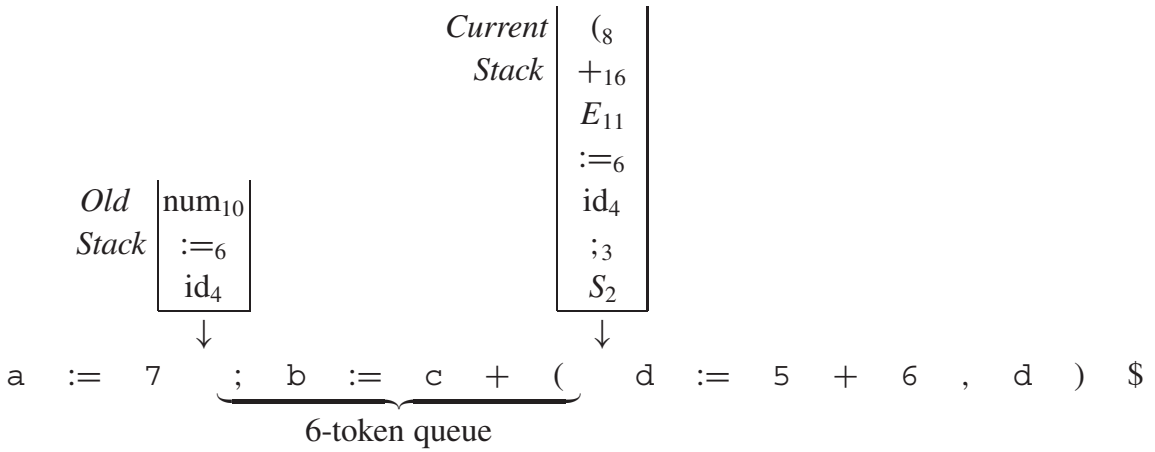


FIGURE 3.39. Burke-Fisher parsing, with an error-repair queue. Figure 3.18 shows the complete parse of this string according to Table 3.19.

engine gets stuck at the 100th token of the input, then it will try every possible repair between the 85th and 100th tokens.

The correction that allows the parser to parse furthest past the original reported error is taken as the best error repair. Thus, if a single-token substitution of `var` for `type` at the 98th token allows the parsing engine to proceed past the 104th token without getting stuck, this repair is a successful one. Generally, if a repair carries the parser $R = 4$ tokens beyond where it originally got stuck, this is "good enough."

The advantage of this technique is that the LL(k) or LR(k) (or LALR, etc.) grammar is not modified at all (no *error* productions), nor are the parsing tables modified. Only the parsing engine, which interprets the parsing tables, is modified.

The parsing engine must be able to back up K tokens and reparse. To do this, it needs to remember what the parse stack looked like K tokens ago. Therefore, the algorithm maintains *two* parse stacks: the *current* stack and the *old* stack. A queue of K tokens is kept; as each new token is shifted, it is pushed on the *current* stack and also put onto the tail of the queue; simultaneously, the head of the queue is removed and shifted onto the *old* stack. With each *shift* onto the old or current stack, the appropriate reduce actions are also performed. Figure 3.39 illustrates the two stacks and queue.

Now suppose a syntax error is detected at the *current* token. For each possible insertion, deletion, or substitution of a token at any position of the queue, the Burke-Fisher error repairer makes that change to within (a copy of) the

queue, then attempts to reparse from the *old* stack. The success of a modification is in how many tokens past the *current* token can be parsed; generally, if three or four new tokens can be parsed, this is considered a completely successful repair.

In a language with N kinds of tokens, there are $K + K \cdot N + K \cdot N$ possible deletions, insertions, and substitutions within the K-token window. Trying this many repairs is not very costly, especially considering that it happens only when a syntax error is discovered, not during ordinary parsing.

Semantic actions. Shift and reduce actions are tried repeatedly and discarded during the search for the best error repair. Parser generators usually perform programmer-specified semantic actions along with each reduce action, but the programmer does not expect that these actions will be performed repeatedly and discarded – they may have side effects. Therefore, a Burke-Fisher parser does not execute any of the semantic actions as reductions are performed on the *current* stack, but waits until the same reductions are performed (permanently) on the *old* stack.

This means that the lexical analyzer may be up to $K + R$ tokens ahead of the point to which semantic actions have been performed. If semantic actions affect lexical analysis – as they do in C, compiling the `typedef` feature – this can be a problem with the Burke-Fisher approach. For languages with a pure context-free grammar approach to syntax, the delay of semantic actions poses no problem.

Semantic values for insertions. In repairing an error by insertion, the parser needs to provide a semantic value for each token it inserts, so that semantic actions can be performed as if the token had come from the lexical analyzer. For punctuation tokens no value is necessary, but when tokens such as numbers or identifiers must be inserted, where can the value come from? The ML-Yacc parser generator, which uses Burke-Fischer error correction, has a `%value` directive, allowing the programmer to specify what value should be used when inserting each kind of token:

```
%value ID  ("bogus")
%value INT (1)
%value STRING  ("")
```

Programmer-specified substitutions. Some common kinds of errors cannot be repaired by the insertion or deletion of a single token, and sometimes a

particular single-token insertion or substitution is very commonly required and should be tried first. Therefore, in an ML-Yacc grammar specification the programmer can use the `%change` directive to suggest error corrections to be tried first, before the default "delete or insert each possible token" repairs.

```
%change     EQ -> ASSIGN  |  ASSIGN -> EQ
       |   SEMICOLON ELSE -> ELSE    |    -> IN INT END
```

Here the programmer is suggesting that users often write "`; else`" where they mean "`else`" and so on. These particular error corrections are often useful in parsing the ML programming language.

The insertion of `in 0 end` is a particularly important kind of correction, known as a *scope closer*. Programs commonly have extra left parentheses or right parentheses, or extra left or right brackets, and so on. In ML, another kind of nesting construct is `let` ··· `in` ··· `end`. If the programmer forgets to close a scope that was opened by a left parenthesis, then the automatic single-token insertion heuristic can close this scope where necessary. But to close a `let` scope requires the insertion of three tokens, which will not be done automatically unless the compiler-writer has suggested "change *nothing* to `in 0 end`" as illustrated in the `%change` command above.

PROGRAM PARSING

Use JavaCC or SableCC to implement a parser for the MiniJava language. Do it by extending the specification from the corresponding exercise in the previous chapter. Appendix A describes the syntax of MiniJava.

FURTHER READING

Conway [1963] describes a predictive (recursive-descent) parser, with a notion of FIRST sets and left-factoring. LL(k) parsing theory was formalized by Lewis and Stearns [1968].

LR(k) parsing was developed by Knuth [1965]; the SLR and LALR techniques by DeRemer [1971]; LALR(1) parsing was popularized by the development and distribution of Yacc [Johnson 1975] (which was not the first parser generator, or "compiler-compiler," as can be seen from the title of the cited paper).

Figure 3.29 summarizes many theorems on subset relations between grammar classes. Heilbrunner [1981] shows proofs of several of these theorems, including LL(k) $\subset$ LR(k) and LL(1) $\not\subset$ LALR(1) (see Exercise 3.14). Backhouse [1979] is a good introduction to theoretical aspects of LL and LR parsing.

Aho et al. [1975] showed how deterministic LL or LR parsing engines can handle ambiguous grammars, with ambiguities resolved by precedence directives (as described in Section 3.4).

Burke and Fisher [1987] invented the error-repair tactic that keeps a K-token queue and two parse stacks.

EXERCISES

3.1 Translate each of these regular expressions into a context-free grammar.

a. $((xy^*x)|(yx^*y))?$

b. $((0|1)^+$`"."`$(0|1)^*)|((0|1)^*$`"."`$(0|1)^+)$

***3.2** Write a grammar for English sentences using the words

```
time, arrow, banana, flies, like, a, an, the, fruit
```

and the semicolon. Be sure to include all the senses (noun, verb, etc.) of each word. Then show that this grammar is ambiguous by exhibiting more than one parse tree for "time flies like an arrow; fruit flies like a banana."

3.3 Write an unambiguous grammar for each of the following languages. **Hint:** One way of verifying that a grammar is unambiguous is to run it through Yacc and get no conflicts.

a. Palindromes over the alphabet $\{a, b\}$ (strings that are the same backward and forward).

b. Strings that match the regular expression $a*b*$ and have more a's than b's.

c. Balanced parentheses and square brackets. Example: `([[](()[()][])])`

*d. Balanced parentheses and brackets, where a closing bracket also closes any outstanding open parentheses (up to the previous open bracket). Example: `[([](()[()][])]`. **Hint:** First, make the language of balanced parentheses and brackets, where extra open parentheses are allowed; then make sure this nonterminal must appear within brackets.

e. All subsets and permutations (without repetition) of the keywords `public final static synchronized transient`. (Then comment on how best to handle this situation in a real compiler.)

f. Statement blocks in Pascal or ML where the semicolons *separate* the statements:

```
( statement ; ( statement ; statement ) ; statement )
```

g. Statement blocks in C where the semicolons *terminate* the statements:

```
{ expression; { expression; expression; } expression; }
```

3.4 Write a grammar that accepts the same language as Grammar 3.1, but that is suitable for LL(1) parsing. That is, eliminate the ambiguity, eliminate the left recursion, and (if necessary) left-factor.

3.5 Find nullable, FIRST, and FOLLOW sets for this grammar; then construct the LL(1) parsing table.

0 $S' \rightarrow S\ \$$

1 $S \rightarrow$
2 $S \rightarrow X\ S$

3 $B \rightarrow$ \ `begin` { `WORD` }
4 $E \rightarrow$ \ `end` { `WORD` }

5 $X \rightarrow B\ S\ E$
6 $X \rightarrow$ { S }
7 $X \rightarrow$ `WORD`
8 $X \rightarrow$ `begin`
9 $X \rightarrow$ `end`
10 $X \rightarrow$ \ `WORD`

3.6 a. Calculate nullable, FIRST, and FOLLOW for this grammar:

$S \rightarrow u\ B\ D\ z$
$B \rightarrow B\ v$
$B \rightarrow w$
$D \rightarrow E\ F$
$E \rightarrow y$
$E \rightarrow$
$F \rightarrow x$
$F \rightarrow$

b. Construct the LL(1) parsing table.

c. Give evidence that this grammar is not LL(1).

d. Modify the grammar **as little as possible** to make an LL(1) grammar that accepts the same language.

***3.7** a. Left-factor this grammar.

0 $S \rightarrow G\ \$$
1 $G \rightarrow P$
2 $G \rightarrow P\ G$

3 $P \rightarrow \text{id}\ :\ R$
4 $R \rightarrow$
5 $R \rightarrow \text{id}\ R$

b. Show that the resulting grammar is LL(2). You can do this by constructing FIRST sets (etc.) containing two-symbol strings; but it is simpler to construct an LL(1) parsing table and then argue convincingly that any conflicts can be resolved by looking ahead one more symbol.

c. Show how the `tok` variable and `advance` function should be altered for recursive-descent parsing with two-symbol lookahead.

d. Use the grammar class hierarchy (Figure 3.29) to show that the (left-factored) grammar is LR(2).

e. Prove that no string has two parse trees according to this (left-factored) grammar.

3.8 Make up a tiny grammar containing left recursion, and use it to demonstrate that left recursion is not a problem for LR parsing. Then show a small example comparing growth of the LR parse stack with left recursion versus right recursion.

3.9 Diagram the LR(0) states for Grammar 3.26, build the SLR parsing table, and identify the conflicts.

3.10 Diagram the LR(1) states for the grammar of Exercise 3.7 (without left-factoring), and construct the LR(1) parsing table. Indicate clearly any conflicts.

3.11 Construct the LR(0) states for this grammar, and then determine whether it is an SLR grammar.

0 $S \rightarrow B\ \$$

1 $B \rightarrow \text{id}\ P$
2 $B \rightarrow \text{id}\ (\ E\ ]$

3 $P \rightarrow$
4 $P \rightarrow (\ E\ )$

5 $E \rightarrow B$
6 $E \rightarrow B\ ,\ E$

3.12 a. Build the LR(0) DFA for this grammar:

0 $S \rightarrow E\ \$$

1 $E \rightarrow \text{id}$
2 $E \rightarrow \text{id}\ (\ E\ )$
3 $E \rightarrow E\ +\ \text{id}$

b. Is this an LR(0) grammar? Give evidence.

c. Is this an SLR grammar? Give evidence.

d. Is this an LR(1) grammar? Give evidence.

3.13 Show that this grammar is LALR(1) but not SLR:

0 $S \rightarrow X\ \$$
1 $X \rightarrow M\ a$
2 $X \rightarrow b\ M\ c$
3 $X \rightarrow d\ c$
4 $X \rightarrow b\ d\ a$
5 $M \rightarrow d$

3.14 Show that this grammar is LL(1) but not LALR(1):

1 $S \rightarrow (X$
2 $S \rightarrow E]$
3 $S \rightarrow F)$
4 $X \rightarrow E)$
5 $X \rightarrow F]$
6 $E \rightarrow A$
7 $F \rightarrow A$
8 $A \rightarrow$

***3.15** Feed this grammar to Yacc; from the output description file, construct the LALR(1) parsing table for this grammar, with duplicate entries where there are conflicts. For each conflict, show whether shifting or reducing should be chosen so that the different kinds of expressions have "conventional" precedence. Then show the Yacc-style precedence directives that resolve the conflicts this way.

0 $S \rightarrow E \$$

1 $E \rightarrow$ while E do E
2 $E \rightarrow$ id $:= E$
3 $E \rightarrow E + E$
4 $E \rightarrow$ id

***3.16** Explain how to resolve the conflicts in this grammar, using precedence directives, or grammar transformations, or both. Use Yacc or SableCC as a tool in your investigations, if you like.

1 $E \rightarrow$ id
2 $E \rightarrow E\ B\ E$
3 $B \rightarrow +$
4 $B \rightarrow -$
5 $B \rightarrow \times$
6 $B \rightarrow /$

***3.17** Prove that Grammar 3.8 cannot generate parse trees of the form shown in Figure 3.9. **Hint:** What nonterminals could possibly be where the $?X$ is shown? What does that tell us about what could be where the $?Y$ is shown?